The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems

Gerard Le Lann

▶ **To cite this version:**

Gerard Le Lann. The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. [Research Report] RR-3079, INRIA. 1996. ⟨inria-00073613⟩

HAL Id: inria-00073613

https://inria.hal.science/inria-00073613v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems

Gérard Le Lann

N° 3079

Décembre 1996

THEME 1

Réseaux et Systèmes

ISSN 0249-6399

The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems

Gérard Le Lann*

Thème 1 - Réseaux et Systèmes

Projet Reflecs

Rapport de recherche n˚3079 - Décembre 1996

26 pages

Abstract: The report issued by the Inquiry Board in charge of inspecting the Ariane 5 flight 501 failure concludes that causes of the failure are rooted into poor S/W Engineering practice. From the failure scenario described in the Inquiry Board report, it is possible to infer what, in our view, are the real causes of the 501 failure. We develop arguments to demonstrate that the real causes of the 501 failure are neither S/W specification errors nor S/W design errors. Real causes of the failure are faults in the capture of the overall Ariane 5 application/environment requirements, and faults in the design and the dimensioning of the Ariane 5 on-board computing system. These faults result from not following a rigorous System Engineering approach, such as applying a proof-based System Engineering method. What is proof-based System Engineering for Computing Systems is also briefly presented.

Key-words: Ariane 5, spaceborne computing system, embedded system, fault, error, failure, method for the engineering of computing systems, real-time system, software engineering, system engineering, user requirements capture, computing system design, computing system dimensioning, design proof, dimensioning proof.

DISCLAIMER

This analysis is meant to -- hopefully -- help those partners in charge of and involved in the Ariane 5 programme. System engineers cannot be “blamed” for not having applied a proof-based System Engineering method, given that it is only recently that such methods have emerged. This analysis is also meant to -- hopefully -- explain why it is inappropriate to “blame” S/W engineers.

(Résumé : tsvp)

* Email: Gerard.Le_Lann@inria.fr
Tel: +33.1.39.63.53.64 - Fax: +33.1.39.63.58.92

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 LE CHESNAY Cedex (France)
Téléphone : +33.1.39.63.55.11 - Télécopie : +33.1.39.63.53.30

La défaillance du vol 501 d'Ariane 5 - Une étude de cas de Génie Système en Informatique

Résumé : le rapport de la Commission d'Enquête constituée à la suite de l'échec du vol de qualification 501 du lanceur Ariane 5 attribue les causes de cette défaillance à des erreurs de Génie Logiciel. A partir du scénario de défaillance décrit dans le rapport officiel, il est possible d'identifier ce que sont, d'après nous, les véritables causes de l'échec du vol 501. On présente des arguments destinés à démontrer que les causes ne sont ni des erreurs de spécification du logiciel, ni des erreurs de conception du logiciel. Les véritables causes de la défaillance sont des fautes de capture des besoins applicatifs et des hypothèses d'environnement relatifs à Ariane 5, ainsi que des fautes de conception et de dimensionnement du système informatique embarqué à bord d'Ariane 5. L'existence de ces fautes provient d'un manque de méthode rigoureuse en Génie Système comme, par exemple, l'absence de méthode imposant des obligations de preuves. On présente brièvement ce qu'est le Génie Système prouvable pour les systèmes informatiques.

Mots-clé : Ariane 5, système embarqué spatial, système embarqué, faute, erreur, défaillance, méthode d'ingénierie système en informatique, système temps réel, génie logiciel, génie système, capture des besoins applicatifs, conception de système, dimensionnement de système, preuve de conception, preuve de dimensionnement.

1. INTRODUCTION

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure, entailing a loss in the order of 1.9 Billion French Francs (~ 0.37 Billion US $) and a 1-year delay for the Ariane 5 programme. An Inquiry Board was asked to identify the causes of the failure. Conclusions of the Inquiry Board were issued on 19 July 1996, as a public report [1]. The failure analysis given in this paper is based upon the Inquiry Board findings. Our conclusions deviate significantly from the Inquiry Board findings. Essentially, the Inquiry Board concludes that poor S/W Engineering practice is the culprit, whereas we argue the 501 failure results from poor System Engineering practice.

This case study is believed to be useful in illustrating the following facts :

(i) confusion between System Engineering and S/W Engineering should come to an end,

(ii) belief that S/W trustworthiness is the most important challenge currently faced by the Computing industry need be re-assessed,

(iii) current industrial practice vis-à-vis System Engineering for Computing Systems is much more empirical than current S/W Engineering practice,

(iv) reliance upon empirical or semi-empirical System Engineering approaches to specifications of Computing Systems is the main reason why so many operational failures and/or project delays and/or cancellations have occurred over the last ten years or so,

(v) the advent of proof-based System Engineering methods for Computing Systems is one of the great challenges set to our profession for the coming years.

The paper is structured as follows. Section 2 introduces our view of those phases of a computing system lifecycle that pertain to System Engineering. In Section 3, we investigate how some of these phases have been conducted in the case of Ariane 5 (according to the findings published by the Inquiry Board) and we identify what we believe are the real causes of the failure. In section 4, we present a detailed review of the Inquiry Board Report, useful to those readers interested in tracing down those diagnostics from the Inquiry Board which have caught our attention. Conclusions are summarized in section 5.

2. PROLEGOMENA

2.1. What is System Engineering of Computing Systems ?

Following the terminology and the definitions of the IEEE Computer Society, the Engineering of Computer-Based Systems (CBSs) is the engineering discipline that encompasses all the engineering disciplines involved in the lifecycle of CBSs, from

the initial expression of end users/clients requirements up to the maintenance and evolution of deployed CBSs.

Consequently, the Engineering of Computer-Based Systems encompasses, e.g., Human-System Interface Engineering, **System Engineering**, Electrical Engineering, S/W Engineering.

System Engineering considers CBSs "in the large". A computing system (CS) is an essential subsystem of a CBS. Hence, System Engineering for CSs considers CSs "in the large" as well, in much the same way Civil Engineers consider bridges or dams "in the large", drawing plans first, then checking they are correct, before starting construction. System Engineering (of CSs) is concerned with, e.g., external event arrival models, architectures, protocols, algorithms, computational models, failure models, redundant constructs, communications.

It is our view that System Engineering main purpose is, for any given project, to produce a global implementation specification of a CS, along with predictions about the CS global behavior, and verification that predicted global behavior matches the initial expression of end users/clients requirements.

A final global implementation specification of a CS usually is modular, each module being known to be implementable (e.g., over commercial off-the-shelf technology, or via specific H/W or/and S/W developments). A module may be implemented as pure S/W + data (e.g., an application program), pure H/W + data (e.g., a databus coprocessor), or as a mixture of S/W and H/W + data (e.g., a "smart" sensor). To be executable, pure S/W modules need be mapped over modules that include some sort of H/W. Global behavior of a CS cannot be predicted unless the H/W factor, the data factor, and modules' interactions are explicitly taken into account.

However, System Engineering is not concerned with how to implement individual modules. System Engineering serves the purpose of splitting some possibly complex initial problem into a number of simpler sub-problems, in such a way that a solution is known to exist for every sub-problem, i.e. it is implementable as a module (some combination of H/W, S/W and data).

Stated differently, how to derive a (modular) implementation specification of a (modular) computing system from the initial expression of some possibly complex application problem falls within the scope of System Engineering. Solving the sub-problems -- i.e. conducting a correct implementation of the corresponding modules -- is the province of Electrical Engineering, S/W Engineering, Device Interface Engineering, Optical Engineering, etc. Implementing a CS module involves specification and design work as well. The crucial observation is that the initial specification of a module cannot be guessed by those in charge of

implementing it. **Before deciding on how a module is going to be implemented, and then apply relevant Engineering methods (e.g., S/W Engineering), it must be the case that a complete and unambiguous specification of that module is available.**

Methods for System Engineering may be more or less "demanding" vis-à-vis formalization or proofs of correctness. For example, specifications may be expressed in human language, in formalized notations, or may result from applying a formal method. In this paper, "specification" is used to refer to any complete and unambiguous statement of a Computer Science problem. We consider that the distinctive attribute of any "good" System Engineering method is the existence of the requirement that proof obligations must be fulfilled, notably the proofs that if every single sub-problem is correctly solved, then the initial (possibly complex) problem is solved as well.

One of the main objectives of the IEEE Computer Society's Technical Committee on the Engineering of Computer-Based Systems [2], as well as one of the goals pursued by the INternational COuncil on Systems Engineering (INCOSE), is the emergence of rigorous (such as proof-based) System Engineering methods for CBSs and for CSs.

2.2. What is proof-based System Engineering for Computing Systems ?

SpECS is the acronym used to refer to provable System Engineering for Computing Systems. Desired global behavior of a CS, as it should be observed by the CS' users and its environment, is specified by the end client/user, via a requirements description (not a specification). Requirements encompass application services as well as the application's environment. Such a description can be viewed as comprising two subsets: one subset $<\Omega, \omega>$ which describes possibly unvalued application requirements and environmental assumptions (e.g., type of timeliness requirements is "latest deadlines for application tasks termination", type of overall dependability requirement is "ultra-high availability" [3], types of arrival laws for external events, types of failures that can be caused by the environment), and another separate subset $<\Omega', \omega'>$ which provides valuations of those unvalued variables appearing in $<\Omega, \omega>$ (e.g., actual values of the latest deadlines for every application task, overall unavailability $< 10^{-9}$, actual values of periods, of arrival densities, highest numbers of failures per mission).

SpECS covers four important phases of a CS lifecycle. For the sake of conciseness, we will not present the phase concerned with how to cope with changes of user/client initial requirements after a CS has been fielded, or with technological upgrades (existing modules replaced with newer modules). In fact, this phase builds upon the

three phases which are examined below. **The three phases of a CS lifecycle that fall within the scope of proof-based System Engineering and which, in a CS lifecycle, precede phases covered by other Engineering disciplines (e.g., S/W Engineering), are as follows:**

- Capture of initial requirements : is concerned with the translation of <Ω, ω> (the possibly unvalued requirements description) into a complete and unambiguous specification of a Computer Science problem, denoted problem <Λ , λ> (possibly unvalued as well); similarly, when available (see further), subset <Ω', ω'> is translated into <Φ , φ>, which provides valuations of those unvalued variables appearing in <Λ , λ>.

- System design : covers all the design stages needed to arrive at a modular (possibly unvalued) specification of a CS, the completion of each design stage being conditioned on fulfilling correctness proof obligations. For example, if "mutual exclusion" is a requirement at some design stage, then that design cannot be proven correct unless it includes a mutual exclusion algorithm that matches those assumptions considered at that stage. Similarly, if latest deadlines are to be met by task terminations, one fulfills a timeliness proof obligation by providing: (i) the expression of a computable function B(m, r), which is an upper bound on response times for any task ranked r-th in module m's waiting queue, r being shown to be the highest rank reachable by that task, (ii) computable feasibility conditions under which function B holds system-wide.

By the virtue of the uninterrupted chain of proofs (that designs are correct), final (possibly unvalued) system design <D, d> (of a CS) -- aggregation of all designs -- provably solves problem <Λ , λ>.

- System dimensioning : covers a single stage that has subset <Φ , φ> (derived from <Ω', ω'>) as an input. The output is a valuation of <D, d>, denoted <V, v>, which is a valuation of those implementation variables that appear in <D, d> (e.g., sizes of memory banks, sizes of data structures, processors speeds, databuses throughputs, number of databuses, processors redundancy degrees, total number of processors, etc.). Many different subsets <Ω', ω'> (hence various subsets <Φ , φ>) may be contemplated by a client/user.

Pair <<D, d>, <V, v>> is the implementation specification of a CS that provably solves problem <<Λ , λ>, <Φ , φ>>.

In other words, capture of initial requirements yields the expression of a generic (possibly unvalued) problem in Computer Science, including assumptions. System design is the resolution of that generic problem, i.e. the expression of a generic (possibly unvalued) solution -- exactly like in Mathematics where one demonstrates

that some theorem holds for every possible value of some variable x, for some axiomatic.

After this is done, the end user/client is free to dimension his/her generic problem, as many times as desired. For each problem dimensioning, there exists a matching dimensioning of the generic solution (i.e., of the CS). To pursue the analogy with Mathematics, the theorem is applied for various values of x. At some point in time, the end user/client decides on some particular problem dimensioning. The specification of the matching dimensioned solution is the implementation specification of a CS. S/W engineers, Electrical engineers, Optical engineers, Telecommunications engineers, etc. can then be put to work, in order to correctly implement the CS as specified.

A major benefit of applying a SpECS method is design reusability. Whenever a new application problem $<\Omega, \omega>^*$ is being considered, and then captured as some specified problem $<\Lambda , \lambda>^*$, it is possible to check whether a problem $<\Lambda , \lambda>$, which is at least as strong as problem $<\Lambda , \lambda>^*$, has been solved in the past. If such is the case, then $<\Lambda , \lambda>^*$ is provably correctly solved with existing design <D, d>, which comes for free.

It is common practice to assign different pieces of an implementation specification to different teams (sub-contractors -- competitors sometimes, in-house engineers). Another major benefit of applying a SpECS method is to get rid of three serious problems arising under current practice, which are as follows:

* specifications of some (a few, many) modules are missing, or are ambiguous, or are incomplete; implementors of modules (H/W or/and S/W engineers) proceed by guess-work, which results into errors that, usually, are not caught before the real system is put into operation,

* verification that the set of concatenated modules "behaves correctly" indeed -- the "system integration phase" -- is a combinatorial problem,

* that verification is done via testing, within some arbitrarily specified amount of time, which in general is short enough to be “acceptable”; which means that, most often, testing is incomplete.

SpECS approaches have the major virtue of requiring (as well as making it feasible) that both the overall design and the dimensioning of a CS are proven correct before starting implementing that CS. It is proven beforehand that the reunion of solutions (of modules) is a global solution. Therefore, if every module is correctly implemented, there is no need to check their interactions. The "system integration phase" vanishes (theoretical viewpoint) or is vastly simplified (practical

viewpoint). Theories and scientific disciplines/techniques which are of interest for SpECS approaches depend on the type of problem <Λ , λ> under consideration, as well as on the design approach adopted to solve it. For example, Queueing Theory would be applied in the case where <Λ , λ>'s type is probabilistic and <Λ , λ> is solved under a probabilistic design approach, whereas Matrix Calculus in (max, +) Algebra would be resorted to for solving a deterministic problem under a deterministic design approach.

Any SpECS method makes it mandatory, at every single design stage, to express/ provide Models, Algorithms and Proofs of Properties (MAPs), such as, e.g., computational Models, failure Models, concurrency control Algorithms, error masking Algorithms, deadline-driven scheduling Algorithms, Proofs by contradiction, Properties by resorting to worst-case analysis and adversary arguments. A method based on a MAPs_for_SpECS approach to deterministic problems has been developed and used by INRIA's project REFLECS for addressing those System Engineering issues that arise with Real-time (R) or/and Distributed (D) or/and Fault-tolerant (F) applications and systems [4, 5]. That method (the TRDF method) has been applied so far to Modular Avionics [6], to Airplane Landing subsystems [7], and to Safety Systems for Nuclear Power Plants [8]. In 1997 - 98, it will be applied by a consortium of six partners to address the generic Real-Time Distributed Consensus problem.

MAPs_for_SpECS methods build upon state-of-the-art in Computer Science. See [9], [10] and [11] for examples of generic problems and solutions.

Many CBS/CS failures and project delays/cancellations have occurred over the last ten years. Examples are TAURUS (Stock Exchange), AAS (Air Traffic Control), SOCRATE (On-Line Reservation), CONFIRM (On-Line Reservation), FREEDOM On-Board Computing Complex (Manned Orbital Station), HORIZON (Shipborne Defense systems), P20 (Nuclear Power Plants), ARIANE 5 Flight 501 (Satellite Launcher).

Analysis reveals that the dominant cause of such failures/delays/cancellations is neither poor project management nor poor S/W Engineering practice, as is often believed, but lack of applying SpECS methods, which results into faulty capture of initial requirements, or system design faults, or system dimensioning faults, or any combination of the above.

Let us now turn our attention to the failure of the Ariane 5 flight 501, which is believed to be an interesting case study in System Engineering for Computing Systems.

3. AUTOPSY OF THE 501 FAILURE

3.1. The failure scenario

The 501 failure scenario described in the Inquiry Board report is as follows. The Ariane 5 launcher started to disintegrate 39 seconds after lift-off, because of an angle of attack of more than 20 degrees, which was caused by full nozzle deflections of the solid boosters and the Vulcain main engine. These deflections were commanded by the On-Board Computer (OBC) S/W, whose input is data transmitted by the active Inertial Reference System (SRI 2). Part of these data did not contain proper flight data, but showed a diagnostic bit pattern of the SRI 2 computer, which was interpreted as regular flight data by the OBC. Diagnostic was issued by the SRI 2 computer as a result of a S/W exception. The OBC could not switch to the backup SRI 1 because that computer had ceased functioning 72 ms earlier, for the same reason as SRI 2.

The SRI S/W exception was raised during a conversion from a 64-bit floating point number F to a 16-bit signed integer number. F had a value greater than what can be represented by a 16-bit signed integer, which caused an Operand Error (data conversion -- in Ada code -- was not protected, for the reason that a maximum workload target of 80% had been set for the SRI computer).

More precisely, the Operand Error was due to a high value of an alignment function result called BH, Horizontal Bias, related to the horizontal velocity of the launcher. The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4, which results in considerably higher horizontal velocity values. The Operand Error occurred while running the alignment task. This task serves a particular purpose after lift-off with Ariane 4 but is useless after lift-off in the case of Ariane 5.

According to the Inquiry Board, causes of the 501 failure are S/W specification and S/W design errors. We now present those faults which, in our view, are the real causes of the 501 failure. Pointers C# appearing in this section refer to comment numbers of section 4.

3.2. Capture of initial requirements/assumptions

Under a SpECS approach, a complete unambiguous statement of problem $<\Lambda , \lambda>$ involves establishing a specification of the following (in particular) :

* set of external events (modeling of the CS environment)
* for every external event :
    - list of every (possibly one) application-level task that is activated
    - arrival law (periodic, sporadic, aperiodic, arbitrary)

* for every application-level task:
  - external variables shared by this task and the CS environment, as well as variables that depend on those external variables **(rc1)**
  - application-level conditions under which that task can/should be activated, suspended, resumed, aborted **(rc2)**,
  - timeliness constraints (latest termination deadline, bounded termination jitter, etc.)

3.2.1. Fact

Horizontal Velocity is an external variable, shared by the environment and some of the SRI module tasks. Horizontal Velocity is used to compute the value of an integer variable called BH. This value, which is sent to the OBC modules, determines commands applied to the nozzle deflections (solid boosters, Vulcain main engine).

Fault RC1 (external variable) :
The need to split the initial description of application requirements into two subsets <Ω, ω> and <Ω', ω'> has not been identified. A fortiori, the need to transform description <Ω, ω> into specification <Λ, λ> has not been identified either. Hence, problem <Λ, λ> has not been stated. Consequently, **external variables, such as Horizontal Velocity, and related variables, such as BH, have not all been listed explicitly** (see rc1 above).

This has resulted into system dimensioning fault DIM1 (comments C2, C4 and C12).

3.2.2. Fact

The alignment task was running after lift-off. The "exception condition" (BH overflow) was raised while running this task. It was later acknowledged that there is no need to keep this task running after lift-off in the case of Ariane 5.

Fault RC2 (application task attribute) :
Had problem <Λ , λ> been stated, the alignment task would have been listed, along with its attributes. In particular, **the condition "alignment task should not be running after lift-off" would have been explicitly specified** (see rc2 above). This has not been done.

This has resulted into system design fault DES5. It was simply impossible for those system engineers in charge of designing/selecting the SRI computer task scheduler as well as eligibility scheduling rules to "guess" that the alignment task should have been aborted right after lift-off. (Comment C5).

3.2.3. Fact

Continuous correct SRI service is essential to a correct functioning of the launcher. This has not been the case. More precisely, the probability of not delivering a correct and continuous SRI service should not be higher than the smallest of the accepted probabilities of failure specified for the other subsystems of the launcher.

Fault RC3 (required dependability property and assumptions) :
A specification of problem <Λ , λ> would have stated the following **dependability property DEP** and related assumptions:

for some **SRI module failure model**, SRI service is correctly and continuously delivered for the entire duration of a flight, with a probability at least equal to 1 - p ■

In other words, up to f SRI modules -- out of n -- "can" fail during a flight. The probability that more than f such failures can occur should be shown to be p at most. Rather, it was **implicitly assumed** that SRI **computers** (H/W only, unlike modules, which are some combination of H/W, S/W and data) would behave according to **"crash-only" semantics.** It was also **implicitly assumed** that **1 SRI computer at most would fail.** This has resulted into system design faults DES1, DES2, DES3 and DES4, as well as into system dimensioning fault DIM2. (Comments C1, C4, C6, C8, C9, C10, C12).

3.2.4. Summary

Beyond the fact that, via the reading of the Inquiry Board report, it is possible to diagnose the three faults given above, one may wonder whether the current Ariane 5 on-board CS is plagued with design/dimensioning faults other than those identified in this paper -- see further, some of them being ascribable to RC faults other than faults RC1, RC2, RC3. Problem is that conclusions and recommendations listed in the Inquiry Board report focus almost exclusively on S/W Engineering issues. It is more than obvious that the expertise required to identify RC faults (fault RC3 in particular) and system design/dimensioning faults is not related to S/W Engineering culture in any respect. Issues raised are System Engineering issues.

The current Ariane 5 on-board CS is a solution to some valued unspecified problem <<Λ , λ>*, <Φ , φ>*>. Reverse System Engineering should be applied in order to identify this problem. Furthermore, <<Λ , λ>, <Φ , φ>> not being stated, it is impossible to compare <<Λ , λ>, <Φ , φ>> and <<Λ , λ>*, <Φ , φ>*>. Portions of problem <Λ , λ>, such as property DEP, can be inferred. Consequently, related design/dimensioning faults are identifiable (see further). Such faults can be corrected. However, doing this does not suffice. **As long as problem <Λ , λ> will not be completely and unambiguously specified, it will be impossible to assert**

that the Ariane 5 on-board CS is correct. Even if flights 502 and 503 turn out to be successful.

Note: no SpECS method was applied either with Ariane 4. It was experimentally "verified" in the course of real launches that potential faults in requirements capture and/or system design/dimensioning/implementation were ironed out (some faults having possibly contributed to a few early failures). Such "experimental" approaches may yield a "confidence level" comparable to that reached with proving, albeit in a more costly manner. Now that SpECS approaches have emerged, we should not keep wasting time and money following semi-empirical System Engineering approaches.

3.3. System design

Under a SpECS approach, MAPs must be explicitly provided for every single design stage. For example, failure models should be explicitly stated, and proofs that DEP holds for such failure models should be given, for some feasibility conditions to be established. Let n be the smallest number of SRI modules needed to cope with up to f module failures, for a given failure model. Making such "deterministic" assumptions as f-out-of-n (property DEP), or any similar assumption of probabilistic nature, does not make sense unless it can be demonstrated that there are no faults that can cause n identical failures (the "common mode" failure scenario). Two approaches can be followed:
- **either prove that** requirements capture and system design/dimensioning/ implementation **faults are avoided; then, non-diversified redundancy can be considered,**
- **or skip such proofs** (which has been done); it is then mandatory to consider **diversified redundancy, in order to tolerate potential faults**, with a sufficiently "high" probability. The greater the value of f, the higher the probability.

This is an important observation. Many "fault-tolerant" CSs are implemented via non diversified redundancy, despite the fact that (i) their design is based on f-out-of-n assumptions, (ii) no proofs that system design and system dimensioning are correct are given. This is like building on quicksand, given that **the activation of any System engineering fault is guaranteed to lead to a total CS failure (n failed modules), whatever the respective values of f and n.**

3.3.1. Fact

Requirement that property DEP should hold was violated. Both SRI modules failed.

Fault DES1 (the SRIs) :
No SpECS method was used. Hence, non diversified redundancy is an incorrect solution. (Comments C1, C8, C12).

Other Space missions have failed for the same reason (Mars Observer is a recent example).

3.3.2. Fact

Both SRI modules failed, exhibiting failure behavior stronger than "crash-only", namely "omission" and "erroneous non-malicious behavior in the value domain".

Fault DES2 (the SRIs) :
The implicit "crash-only" assumption has not been specified. Hence, it could be violated, which happened. Detection-and-recovery cannot be proven to be a correct solution unless failure models are specified.

For property DEP to hold, it is well known that active redundancy is the only algorithmic construct that can be considered in the presence of unspecified failure models (i.e., stronger than "crash-only" a priori). It is also well known that $n > 2f$ is a necessary and sufficient feasibility condition under such circumstances. (Comments C1, C7, C8, C10, C12).

Fault DES3 (the SRIs and the OBCs) :
The implicit "crash-only" assumption has not been specified. Hence, rather than "committing suicide", SRI modules issued diagnostic/error messages.

Under a SpECS approach, every event that can be posted to a module must be listed. Similarly, tasks that can be activated over a module must be listed. A {task, event} mapping must be provided. Had this been done, event "flight data" could not be confused with event "exception condition reporting". Consequently, System engineers would have identified the need to provide for (among others) **two separate OBC tasks, one in charge of executing the flight program, the other one in charge of handling exception conditions (error messages).** (Comments C2, C4, C5, C8, C12). Had this been done, "crash-only" behavior would have been correctly implemented vis-à-vis the OBC flight program.

Although this would not have helped avoid the 501 failure, this is a latent fault that may lead to a future flight failure.

3.3.3. Fact

The OBC modules (H/W + S/W + data) in charge of commanding the nozzle deflections interpreted error messages issued by the SRI modules as flight data.

Fault DES4 (the OBCs) :
No proof has been established showing **that a correct OBC module delivers correct inputs to the OBC task** in charge of commanding the nozzle deflections **in the presence of some faulty values issued by the SRI modules.**

A correct OBC module can be shown to compute a correct value out of a set of values issued by n SRI modules, f of them at most being faulty, by using an error masking algorithm (e.g. majority voting). A harder problem, known as the "Byzantine generals" problem, was solved long ago, for the most unrestricted failure model [12]. This work has influenced the design of the US Space Shuttle on-board CS.

In passing, one may observe that this "checking" conducted by the OBC modules would have compensated for the lack of checking (of BH values) by the SRI modules. And the 80% maximum workload target set for SRI computers would not have been exceeded. This has not been done due to lack of having rigorously looked at the on-board CS "in the large". (Comments C1, C3, C5, C7, C11).

3.3.4. Fact

Loss of correct flight data occurred while running the alignment program. With Ariane 5, keeping this program active after lift-off is unnecessary.

Fault DES5 (the SRIs) :
Conditions under which tasks can be run, suspended or aborted are to be stated explicitly so as to specify problem $<\Lambda , \lambda>$ (see rc2, section 3.2).

Had this been done, and assuming a correct task scheduler has been designed for the SRI modules (not a S/W engineering issue), the "alignment" task would have been deactivated (by the task scheduler) as specified, e.g. right after lift-off if so desired. (Comments C5, C12).

3.4. System dimensioning

No rigorous requirements capture having been conducted, and design faults having been made, provably correct dimensionings of some of the variables appearing in the final on-board CS implementation specification were missing (BH and f, in particular). That could not be detected during the (S/W and/or H/W) implementation phase, unless accidentally. This resulted into an incorrect size of the memory space used for storing variable BH. This resulted also into an empirical dimensioning of the SRI modules group.

Recall that one cannot proceed with implementation (e.g., S/W implementation) unless system dimensioning is complete (<V, v> is available -- see section 2).

3.4.1. Fact

The Operand Error was due to an unexpected high value of BH.

Fault DIM1 :
Range of values taken by variable BH was assumed to be in the {- 2^{15}, + 2^{15}} interval.

Had a SpECS method been used, fault RC1 would have been avoided. Hence, the client/user or some representative -- e.g., a space/flight engineer -- would have been explicitly asked to quantify the ranges of possible values of Horizontal Velocity and BH, for the application under consideration, that is Ariane 5 (not Ariane 4). This would have translated into a correct dimensioning of the memory block used to implement BH.

Similar implicit assumptions may have been made also for other variables or data structures.

Again, dimensioning issues are System Engineering issues, not S/W Engineering issues. (Comments C1, C2, C3, C4, C7, C9, C12).

3.4.2. Fact

Both SRI modules failed. The f-out-of-n (1-out-of-2) assumption was violated.

Fault DIM2 :
It was implicitly assumed that f's value would be 1 and that simple redundancy (n = 2) would suffice. No correctness proof (that probability 1 - p would be higher than some quantified value) was given.

The intrinsic reliability of a SRI module (H/W + S/W + data), and related coverage factor, can be derived from statistics computed over accumulated experimental data, or via probabilistic modeling. Intrinsic reliability and coverage factor, combined with launch duration, are used to compute a value for f, as well as related probability q that at most f failures “can” occur. Obviously, the smallest correct value of f to be chosen should be such that 1 - q < p holds true.

It is thereafter possible to derive the lower bound of n which, depending on the (non malicious) failure model that should have been considered, is f+1 or 2f+1.

It appears that System engineers took H/W only into consideration. This has also led to a violation of the n > 2f lower bound, which applies whenever non malicious failure semantics are not specified. (Comments C1, C7, C8, C9, C10, C12).

Although a value of f (resp. n) greater than 1 (resp. 2) would not have helped avoid the 501 failure, this is a latent fault that may lead to a future flight failure.

4. REVIEW OF THE INQUIRY BOARD REPORT

4.1. Detailed review

Page numbering is in reference to the Inquiry Board Report. Comment numbers C# are those used as pointers in section 3.

C1. Page 3"... In order to improve reliability, there is considerable redundancy at the equipment level ..."

Not considerable at all in fact. This is non diversified simple redundancy (degree 2), i.e. the weakest type of redundancy that can be imagined. Many existing systems involve degree 3 (triple) redundancy (e.g. Tandem computers in the marketplace, others in the case of special-purpose applications). Some systems involve degree 4 redundancy. This is the case with redundant dual modules (e.g. Stratus computers in the marketplace, others in e.g. Nuclear Power Plants), or plain quadruple redundancy (e.g. the US Space Shuttle on-board CS, which, in addition, has a 5th spare computer for in-mission repair).

Neither H/W redundancy nor S/W redundancy help if a redundant construct is tainted with System Engineering faults (faults RC3, DES1, DES2, DES3, DES4, DIM2).

C2. Page 4:"... SRI S/W exception ..."

Whether the dimensioning of a physical resource used to implement a variable is a S/W engineering issue or a System engineering issue depends on the variable's type. For every variable that is used in a program, there are two possibilities:
- it is a private variable, i.e. purely internal to that program,
- it is a public variable, i.e. shared by that program and other entities.

A special case of a public variable is an external variable, i.e. a variable which is shared by a computing system (one or many programs) and its environment (e.g., sensor data).

For every private variable, it is the sole responsibility of a S/W engineer to define the range of values taken by that variable, hence the physical dimensioning of the corresponding physical resource.

Conversely, for every external variable, and for any variable that depends on an external variable, range of possible values is derived from constraints or considerations (e.g., Physics) that are not under the control of a S/W engineer.

Therefore, in the latter case, finding the correct size of the physical resource (memory space) used to implement that variable is not a S/W engineering issue, but a dimensioning (of a system resource) issue. Similarly, finding what is a correct buffering capacity for a waiting queue is not a S/W engineering issue, but a dimensioning issue (of that system module that implements the waiting queue).

BH is related to external variable Horizontal Velocity. Hence, the appropriate size of a memory block for storing BH values depends on the horizontal velocity of the launcher. Why should it be that this consideration -- which is pure Physics -- should be treated or viewed as a S/W engineering issue?

What if the reading/conversion process had been implemented in H/W? Would then the concept of horizontal velocity become an Electrical engineering issue? S/W engineers are no more and no less qualified than H/W engineers to grasp the concept of horizontal velocity. In any case, how to conduct a correct dimensioning of an external variable is not under their responsibility.

The reading/conversion procedure used to compute BH values is correct. This procedure is correctly implemented (as "demonstrated" by the many Ariane 4 flights). S/W engineers made no design errors (w.r.t. the 501 failure). They were not/ could not be in charge of deciding on the dimensioning of the "BH memory block".

Calling an incorrect dimensioning of a memory block a "S/W design error" is as erroneous as calling the selection of too slow a processor a "S/W design error". (Faults RC1, DIM1).

C3. Page 5: "... maximum workload target (of a SRI computer) = 80% ..."

Why 80% ? How have such feasibility conditions been established? Which are the proven properties enforced with meeting such conditions? These are questions that would inevitably be asked by a System engineer. The answers would likely be "timeliness properties". Ability to develop proofs that such properties are enforced -- and ability to check the proofs -- are not related at all to S/W engineering culture.

For example, given the dimensioning of an arrival law for a certain external event, it is possible to compute the maximum length of the corresponding waiting queue, hence the exact buffering capacity needed for a correct implementation (i.e., no event is ever lost) of that waiting queue. However, doing this implies in the first place that a System engineer has been able to identify and prove what are the worst-case

scenarios for the pair <arrival law, event scheduling algorithm> under consideration. This is definitely not something one can expect being done by a S/W engineer. Only a System engineer who is experienced in Scheduling theory can do the job. In the case of BH, "worst-case scenario" translates into an upper bound on the range of possible BH values. Furthermore, the OBCs should have been designed so as to cope with the 80% upper limit set for the SRIs. (Faults DES4, DIM1).

C4. Page 5: "Although the failure was due to a systematic S/W design error..."

The failure is due to the existence of multiple System Engineering faults. The SRI computer S/W did no mistake. The design of the SRI computer S/W is not erroneous in regard with the specification at hand. The real problems are:

(i) the specification of the SRI module (not just the S/W) was incomplete (a system design fault w.r.t. fault-tolerant computing),
(ii) buffering provided to store the result of its correct computations was too small (a system dimensioning fault).

Here, we are specifically concerned with problem (ii). Exception mechanisms are used by S/W engineers to refer to "anything else", i.e. unexpected/unanticipated events or states. The set that comprises these events/states may be quite large. Not surprisingly, partial exploration of this set triggers ideas about how to cope with each of those specific events/states under consideration (what is called "mechanisms ... to mitigate ... problem" in the Report). But doing this is not very interesting -- nor efficient.

Another way of showing that the statement "it's a S/W design error" is meaningless is as follows: imagine that, e.g. for the sake of fast execution, the conversion procedure had been implemented in H/W (an ASIC, for example), with the same underdimensioned memory space for BH. Flight 501 would have failed as well. However, the Inquiry Board would have concluded: "it's a H/W design error". That would be equally meaningless. What matters is not whether it's H/W or S/W which runs the correct (fault-free) conversion procedure. What matters is that the thing which runs correctly the correct procedure, which produces the correct results, had not enough room to store its results. (Faults RC1, DIM1).

C5. Page 6:"... presumably based on the view that ... not wise to make changes in S/W which worked well with Ariane 4".

Which S/W modules are part of a computing system, what their constraints are, all this depends on the user/client originated description of application/environment requirements. Under a SpECS approach, the requirements capture would have revealed that those S/W modules and/or their constraints, as inferred from the

characteristics of Ariane 5 flights, are not exactly those identified for Ariane 4. Hence, "the S/W had to be modified". (Faults RC2, DES5).

A major reason for not modifying an existing set of programs often is that "the S/W has been debugged". But what good is it if the "bug-free" S/W does not implement what is needed?

Furthermore, suppressing a few inadequate application programs from an existing set of programs, and adding new application programs to this set, is easily feasible provided that:
(i) the initial overall design of application S/W is based on modularity principles,
(ii) every application program is guaranteed to run as expected, whatever the other application programs are (except those it needs to run to completion).

The second property is known (to System engineers and Computer scientists) as the atomicity/isolation property. It is easily obtained by resorting to a (system-level) algorithm that enforces what is known as serializability [9]. This is an excellent illustration of how S/W engineering problems can be greatly eased by resorting to solutions that have been around for years in the System engineering community. Not taking advantage of such knowledge yields undue S/W inflexibility, and therefore artificially inflated S/W costs (besides failures).

Moreover, it has been known for long that all sorts of obscure run-time errors -- which may result into flight failures -- can arise whenever tasks are multiplexed over a computer, even if every task is individually proven correct. This problem has solutions, instantiated as system-level algorithms, such as, for example, concurrency control, mutual exclusion, deadline-driven scheduling. Obviously, such solutions are not the province of S/W engineering.

Does reluctance to make changes to the Ariane 4 S/W mean that the Ariane 4/Ariane 5 S/W is not modular? Does this mean that it is impossible or too difficult to modify the existing system-level S/W so that change of operational mode right after lift-off would automatically result into having the alignment program prohibited from running?

C6. Page 6:"... S/W is flexible ... thus encourages highly demanding requirements ... complex implementations difficult to assess."

Requirements, hence application/environment complexity, are originated by the end user/client. They are fully and exclusively determined by what it is the user/client wants to be provided with, from a strictly application/environment oriented viewpoint (Ariane 5 flights in our case). At the time the user/client expresses (possibly demanding) requirements, he/she has the faintest idea about what is going

to be implemented in pure H/W, in firmware (to be run over some piece of H/W), in S/W (to be run over some piece of H/W). In the case of Ariane 5, approximately 10 years separate the definition of the launcher missions from implementation of the on-board CS.

Furthermore, whether or not possibly demanding application requirements are met cannot be verified by "assessing" the implemented S/W. (Similarly, Civil engineers do not verify the "correctness" of a bridge simply by "assessing" the implemented concrete. They do check the bridge plans first). An implementation (in S/W, in H/W) is to match a specification, such a specification being obtained by applying a SpECS method. It is the responsibility of System engineers to prove that a global modular specification of an entire computing system meets the user/client originated requirements. Implementation of each individual module comes after. **Complexity of user/client requirements is mostly dealt with by System engineers.**

Would end user/client requirements be considered "less demanding" if it turns out that, thanks to advances in H/W technology, requirements can be met by resorting to pure H/W modules?

C7. Page 6:"... S/W should be assumed to be faulty until ... best practice methods can demonstrate that it is correct."

Yes, definitely so. The same recommendation obviously applies to every discipline involved with the development of a CS. What if the H/W would not be proven (or checked to be) correct? Hence, one should also read: "System design and dimensioning should be assumed to be faulty until ... that they are correct".

It is well known that "best practice" S/W methods -- whether or not based on formal approaches -- have limitations in terms of levels of complexity that are tractable [13]. It is precisely the purpose of a SpECS method to break complexity as much as needed. Another benefit of using a SpECS method is that concatenating the sub-systems (so as to build a global system) does not raise the conventional "system integration problem", which we know is a combinatorial problem under current practice. This explains why, with "critical applications", it does not make sense to rely almost entirely on testing at the integration phase, simply because it takes too long (dozens of years) to achieve a confidence ratio in the order of the reliability/ availability figures specified by such clients/users as ESA.

Testing activities are shrunk in order to match predetermined time and finance budgets that have no relation whatsoever with achieving some very high level of overall confidence. **Flight 501 can in fact be viewed as a -- rather costly -- continuation of an incomplete testing procedure.**

C8. Page 7:"... providing complete test coverage of each sub-system and of the integrated system."

Even if one would assume that complete test coverage is feasible for each sub-system, it would still be the case that, under current practice, complete test coverage of the integrated system cannot be achieved. This is so for the reason that the decomposition of a system into modules (top-down approach) or composition of modules into a system (bottom-up approach) are conducted empirically, i.e. without fulfilling proof obligations. (Faults DES1, DES2, DES3, DES4, DIM2).

C9. Page 7:"It should be noted ... physical law, ... test the SRI ..."

Knowledge needed to conduct tests, as described, exactly is that knowledge needed to apply a SpECS method. The fundamental difference is that, in the latter case, one proves that a system design/dimensioning is correct before implementation, whereas in the former case, one first builds a system, whose implementation is then tested so as to possibly discover that something is wrong. In addition to being (very) costly and time consuming, this leaves open the issue of discriminating between faulty capture of the user/client's requirements, system design faults, system dimensioning faults, H/W errors, S/W design faults, S/W implementation faults, and the like.

C10. Page 8: "... it is logical to replace them with simulators ... scope completely."

It is first of all mandatory to mathematically establish the accuracy or the level of confidence of the simulators, i.e. to establish how close they are to reality. In particular, it is mandatory for their accuracy to be at least comparable to user/client defined levels of overall confidence. In any case, simulation cannot achieve levels of confidence comparable to those attained with proofs. (Faults RC3, DES1, DES2, DES3, DES5, DIM1, DIM2).

C11. Page 9:"... amongst them to prove the proper system integration of equipment such as the SRI."

By simulation? Is it possible to prove that an inherently stochastic approach to the verification of system integration yields the desired high levels of confidence, in the presence of "demanding" user/client requirements?

Simplifying -- doing away, from a theoretical viewpoint -- the system integration testing phase precisely is one of the major benefits of adopting a SpECS approach. It does not seem wise to defer to the testing phase the discovery of potential System engineering faults. Neither is it wise to rely on chance to find them.

C12. Page 12:"This loss of information was due to specification and design errors in the S/W of the inertial reference system."

Note: comments given below apply only to those parts of the S/W that have been deemed erroneous by the Inquiry Board (comments do not apply to other pieces of S/W).

*** "Specification errors in the S/W"**
The specification of the SRI module (not just the S/W) was not erroneous, but incomplete. Furthermore, that is not the real cause of the failure. That is just a manifestation of the real causes, which are faults RC1, RC2, RC3, DES1, DES2, DES3, DES4, DES5.

Again, S/W engineers cannot guess what it is which is missing in a specification, w.r.t. fault-tolerant constructs and redundancy management algorithms. They cannot bear any responsibility for having been given incomplete specifications (e.g., neither crash-only semantics nor diversified redundancy were specified).

*** "Design errors in the S/W"**
The SRI S/W engineers made no S/W design errors (w.r.t. the 501 failure). On the one hand, they produced S/W designs which appear to be correct in regard with those (incomplete) specifications they were given. On the other hand, they designed a correct conversion procedure.

They cannot bear any responsibility for not having been provided with a dimensioned specification (which should have contained the size of the memory space needed to store BH values), which is under the responsibility of a System engineer. (Faults RC1, RC3, DIM1).

It is misleading to label faults in requirements capture and in system dimensioning as S/W design errors.

4.2. Summary

The conclusions of the Ariane 5 flight 501 failure Report have been reviewed. We have found that these conclusions are unjustly directed at S/W engineering errors. This is yet another example of a long lasting confusion between S/W engineering and System engineering. In an attempt to dispel such harmful misconceptions, we have argued that what is called S/W specification errors or S/W design errors are manifestations of more profound causes, namely System engineering faults in requirements capture, system design, and system dimensioning. These faults are the real causes of the 501 failure.

System design and system dimensioning issues have not been inspected by the Inquiry Board. Consequently, it may be that the current Ariane 5 on-board CS is plagued with other system design/dimensioning faults which may lead to future flight failures. Most of them will remain undetected if S/W only is believed to be at stake.

Expertise required to identify -- or to avoid -- System engineering faults is not at all related to S/W engineering culture (nor to H/W engineering culture). Such expertise has to do with algorithmic and architectural constructs, which are essential for achieving, e.g., error compensation, majority voting, distributed synchronization, deadline driven task scheduling, application task atomicity/isolation, distributed agreement, and the like. System engineering expertise is also required to translate some incomplete and/or ambiguous description of application/environment requirements into a complete and unambiguous specification of a Computer Science problem (which, in the case of Ariane 5, would involve ultra-high dependability and strict timeliness properties). Ability to address such issues, and how to develop proofs of correctness, calls for expertise in such areas as Decision Theory, Mathematical Logic, Serializability Theory, Graph Theory, Scheduling Theory, Distributed Algorithms, as well as mastering of such techniques as Adversary Arguments, Worst-Case Analysis, or Matrix Calculus in (max, +) Algebra.

A particularly striking illustration of the different views that result from the fundamental differences existing between System engineering and S/W engineering lies with comment C1. We have identified -- among others -- a serious system design fault, which mirrors knowledge that is reasonably widespread in the System engineering community, namely that any CS based on non diversified redundancy inevitably fails whenever a system design or a system dimensioning fault is activated, whatever the redundancy degree. Obviously, such knowledge -- hence the fault -- was ignored by the Inquiry Board. The Board found no weakness w.r.t. the existing redundant on-board CS, which is demonstrated by the diagnosis that "there is considerable redundancy at the equipment level".

Despite evidence that S/W engineers cannot be blamed for the 501 failure, improving the quality of the S/W design and production process is a reasonable recommendation, as usual. (It is almost always the case that S/W can be improved). However, it is essential to understand that looking at CSs as S/W "things" inevitably leads to disillusions. And to erroneous or inaccurate failure analyses. Let us consider a simple analogy with Medicine. Skin blisters is the observable problem (the "failure"). Skin (the "S/W") is examined by experts, that is dermatologists ("S/W experts"). They recommend different skin treatments, such as antibiotic creams and no sun exposure ("S/W will thus be fixed"). Alas, new blisters keep popping out (repeated "failures"). This lasts until a physician (a "System expert") finds out that

the cause of the problem is repeated food poisoning (a "faulty requirements capture": edible food only should be eaten; or a "faulty system design": human beings should be designed so as to safely eat non edible food).

Expertise in such areas as those listed above is not required to be a good S/W engineer. S/W engineers are not expected to be experts in System engineering, and vice-versa. Different problems call for different disciplines and for different experts.

5. CONCLUSIONS

It is now possible to understand what led to the failure of flight 501. If we were to stick to **internationally accepted terminology**, we should write the following:

Faults have been made while **capturing the** Ariane 5 application/environment **requirements**. **Faults** have also been made in the course of **designing and dimensioning the Ariane 5 on-board CS**. These faults have resulted into **errors at the system modules level.** Some of these **errors have been activated** during launch. **Neither algorithmic nor architectural constructs could cope with such errors. Flight 501 failure ensued.**

Had all the modules -- as they were specified -- been implemented in H/W, flight 501 would have failed the way it did. Had the implementation of every module -- in S/W and/or in H/W -- been proven correct in reference to the specifications at hand, flight 501 would have failed the way it did. Therefore, causes of the failure belong neither to the "S/W world" nor to the "H/W world". Issues at stake are System engineering issues.

Proof-based System engineering eases the job of S/W engineers (faults that would result into S/W errors are avoided). Conversely, proof-based S/W engineering does not compensate for poor System engineering practice. Similarly, in Civil engineering, "good" reinforced concrete cannot compensate for lack of having global plans proven correct in the first place.

The major conclusion is that the capture of initial application requirements, system design and system dimensioning issues, have been vastly overlooked. Whether or not the existing Ariane 5 on-board CS, or any future Ariane 5 on-board CS, is free from design and/or dimensioning faults other than those identified in this paper can only be established by applying a SpECS method.

A fortiori, it is strongly recommended that a SpECS method be applied to other Space programmes.

References

[1] Inquiry Board Report, "ARIANE 5 - Flight 501 Failure", Paris, 19 July 1996, 18 p. [http://www.inria.fr/actualites-fra.html]

[2] IEEE Computer Society
* Task Force on ECBS, "Systems Engineering of Computer-Based Systems", Computer, Nov. 1993, 54-65.
* S. White, J. Rozenblit, B. Melhart, "Engineering of Computer-Based Systems : Current Status and Technical Activities", Computer, June 1995, 100-101.

[3] J. Gray, D.P. Siewiorek, "High-Availability Computer Systems", Computer, Sept. 1991, 39 - 48.

[4] G. Le Lann, "Certifiable Critical Complex Computing Systems", 13th IFIP World Computer Congress, 1994, (K. Duncan, K. Krueger Eds.), (Elsevier Science B.V. Pubs.), vol. 3, 287 - 294.

[5] G. Le Lann, "A Methodology for Designing and Dimensioning Critical Complex Computing Systems", IEEE Intl. Symposium on the Engineering of Computer-Based Systems, Friedrichshafen, Germany, 11-15 March 1996, 332-339.

[6] INRIA Project REFLECS, "Algorithmique TR/TD/TF ORECA", 4 reports, French DoD contract DRET 94 - 395, 1995 - 1996.

[7] INRIA Project REFLECS, “Etude de la Tolérance aux Fautes pour la Fonction Trains-Trappes d’un Système Atterrisseur”, 1 report, contract GENIE (French Ministry of Research), July 1996.

[8] INRIA Project REFLECS, "Méthode TRDF et Sécurité des Centrales Nucléaires", 1 report, contract Institut de Protection et de Sûreté Nucléaire 4040 - 6B024870/SH, September 1996.

[9] P. A. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley Pub., ISBN 0-201-10715-5, 1987, 370 p.

[10] M. Joseph (Editor), "Real-Time Systems - Specification, Verification and Analysis", Prentice Hall UK Pub., ISBN 0-13-455297-0, 1996, 278 p.

[11] N. A. Lynch, "Distributed Algorithms", Morgan Kaufmann Pub., ISBN 1-55860-348-4, 1996, 872 p.

[12] L. Lamport, R. Shostak, M. Pease, "The Byzantine generals problem", ACM Transactions on Programming Languages and Systems, vol. 4, 3, July 1982, 382 - 401.

[13] J. Rushby, "Formal Methods and the Certification of Critical Systems", SRI - CSL Report n 93-07, Nov. 1993, 308 p.

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 1655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur

Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399